intel

TECH
PAPER

TP-2

November 1982

iAPX 286: A Microsystem for the New Generation of Operating System Intensive Applications

George Alexy
Intel Corporation

This paper was prepared for and presented at Wescon®/82.

AFN-00136A
ORDER NUMBER: 210839-001

iAPX 286: A Microsystem for the New Generation of Operating System Intensive Applications

George Alexy
Product Marketing Manager
High-End Processors
Intel
2625 Walsh Road
Santa Clara, CA 95051

Since the introduction of the iAPX 86 16-bit microsystem in 1978, the application of microprocessors has quickly evolved from the dedicated function markets like semi-intelligent ASCII terminals and simple control and instrumentation to more complex applications like multi-terminal fixed task systems (e.g., word processors), single user reprogrammable business systems and process control. This evolution has taken advantage of the new capabilities and performance of the 16-bit micro's to satisfy markets needing more capabilities than 8-bit micro's and lower costs than mini-based products.

This thrust into markets previously requiring low end minicomputers, has developed new requirements for microprocessors. As these markets evolve, system designers increasingly require minicomputer-like functionality for product growth. Migration from single-user, dedicated multi-function to reprogrammable multi-user and multitasking requires features like memory protection and memory management in addition to higher throughput. These features must allow not only protection of the O.S. from users but also users from each other. Effective multi-user system implementations will need virtual memory to be part of the overall memory management strategy to eliminate restrictions on the number of users supported by the system. It is essential that these features be provided without impacting system throughput or system cost (in terms of hardware and software). An additional requirement for these markets is the ability to migrate application software and data bases to new products.

If we look at the system environment of a typical multitasking or multi-user reprogrammable system, we can start to envision the needs of these systems. The system consists of an operating system and one or more tasks or users. The most obvious need for protection in the system is isolation of the operating system and data associated with the operating system from malicious or accidental corruption by the task or user programs. If multiple tasks or multiple users are active in the system, it is also necessary to isolate the individual tasks or users from each other. While the first case is necessary to guarantee the fundamental integrity of the operating system and prevents errant tasks or user programs from disrupting the system, the second case is equally important if individual tasks or users are to truly be provided a secure operating environment isolated from the side effects of other user's erroneous software. As a result, a single task or user's environment should only be affected by the O.S. or the task or user program itself.

A typical system environment goes beyond this simple model. In addition to the O.S. and individual tasks and users, the system must support sharing of code and data. Shared code would include system utilities, special application package libraries like floating point math, sort/merge etc., and runtime interfaces for languages. Shared data must be supported if tasks or users are to communicate with each other. The result is a requirement to provide full protection, yet simultaneously support controlled access to common code and data areas without being hindered by the protection system. This system composition leads to the structure shown in Figure 1.

Figure 1 shows the basic O.S. and task or user communications interfaces. The users or tasks must be able to access shared code and data in addition to O.S. services while otherwise being fully protected from each other. The O.S. on the other hand must be fully protected from erroneous use or access by the users or tasks while maintaining free and efficient access to task or user address space. The later aspect is desirable for responding to valid user requests of O.S. services like memory allocation, message communication and fault handling.

Assuming sufficient intertask and O.S. isolation is provided, the integrity of the system becomes dependent on the integrity of the operating system itself. For relatively

simple monitor oriented operating systems, proving the integrity of a complete O.S. may be feasible; however, as the systems evolve into more complete multitasking and reprogrammable multi-user systems, validating the O.S. integrity becomes an incomplete and inexact activity of trial and error in an evaluation environment. This somewhat questionable technique of product integrity validation has lead to the need for isolation and protection within the O.S. itself. The most critical system-wide O.S. procedures and data, like fault handlers, memory allocations/de-allocation, task dispatching and system state information must be protected from the bulk of system software associated with activities like file management, job control and system debuggers. This approach allows the most critical system functions to operate correctly even if other system software is not fully debugged. It also minimizes the set of software (often referred to as kernel software) that the system absolutely depends on and should be provably error free.

With respect to the model in Figure 1, isolation within the O.S. would be portrayed as a separation of the O.S. into kernel and non-kernel levels where task and user software continue to maintain direct interfaces to both levels of software. If system software is upgraded to accommodate new features, the new software is typically assigned to the non-kernel level where bugs in the new software, while potentially affecting overall system operations, cannot cause catastrophic, unreportable errors. This can significantly enhance the overall integrity of the system.

While the protection and resulting system integrity previously discussed is certainly desirable, it cannot be allowed to degrade overall system performance. This is particularly relevant in multitasking systems where the real time response of the system (e.g., task suspension, context save and lead and initiate of new task) is typically the most important system parameter. In a protected environment this can become a complex operation if protection state information must also be swapped as part of the task state. It also becomes important in multi-user systems to maintain reasonable user response time.

Equally as important in multitasking systems is basic interrupt response time. In many protected systems, a task switch is required to invoke the interrupt service since the interrupt handler is a system wide service not associated with the current user or tasks system context. In many instances this separation is desirable since the interrupt will cause the dispatch or scheduling of a service routine that may not be time critical. However, for interrupts

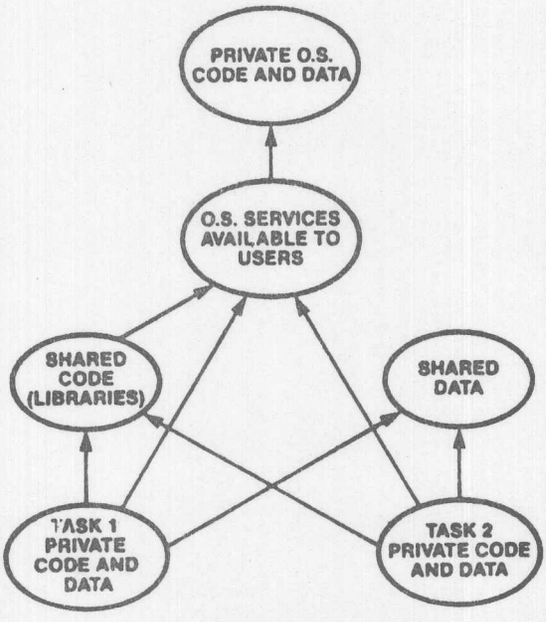


Figure 1. Typical Multi-user, Multitasking System.

that do require immediate attention, the basic system task switch time of hundreds of micro seconds or even milliseconds will be unacceptably long.

In addition to interrupt response time and task or user activation time, the speed with which system services can be accessed by user software also affects the overall system performance. This aspect of performance is critical primarily in multi-user systems where a rich set of extended features associated with the operating system is heavily accessed by user software. In simple non-protected systems, access is implemented by simply pushing appropriate information onto the stack and calling the desired function. In protected systems even this simple access technique becomes relatively complex. User and system software are typically isolated in different address spaces to prevent erroneous access to system data and software by user software. The simple call mechanism is replaced by a special supervisor (O.S.) call instruction that tells the machine to switch address spaces. This forces the user to have specific knowledge about system services and must differentiate them from normal application services with this special call instruction. Even more complex is the parameter passing sequence. The caller places parameters on his stack which is not directly accessable from the system software address space. The system software must gain access to the user's stack and copy the parameters to the system stack or continue to directly use the user's stack. A side affect of this is the potential for user stack errors to cause errors during systems software execution. Improper initialization of the stack by the user software before transferring control to system software can cause stack overflow or underflow to occur during system software access to the user stack, thereby allowing a user error to result in a system error. Depending on the system design, errors that occur within the O.S. are often considered non-recoverable. If this is allowed to happen as a result of user error, the protection in the system will have been circumvented and the intent of providing protection violated. To compensate for this and other similar cases typically requires nontrivial software that increases system complexity and reduces system performance and reliability.

The general separation of address spaces between user and system also has implications on the systems ability to access user data areas such as text buffers, files and I/O buffers. Without appropriate system structure to allow the O.S. access to user address space, software must be written to translate the tansferred addresses to a form usable by the O.S. Although this typically is not critical to system performance, it represents additional complexity when dealing with a protected multi-user or multitasking system.

One last aspect of this next generation of microprocessor based systems, particularly important in multi-user applications is virtual memory. These systems will need to support multiple simlultaneous users whose individual code and data space requirements exceed the physical memory of the system. To allow for software portability, use of high level languages and simplified program implementation, the physical memory of the system should be transparent to the user and not reqire special programming practices like overlays. Support for virtual memory basically requires that all code and data be dynamically relocatable and access to code or data not in physical memory be detectable and recoverable.

A microprocessor designed to address the needs of this next generation of multi-user and multitasking systems is Intel Corp. iAPX 286. The design goals of the processor were specifically targeted at providing very comprehensive memory management and protection while also significantly enhancing the performance of these systems.

We can start to understand the mechanisms of the iAPX 286 and how they accommodate the needs of these systems by first looking at how the user views his virtual address space. Figure 2 shows that the task or user has a one gigabyte virtual address space partitioned into a half gigabyte of global and half gigabyte of local address space.

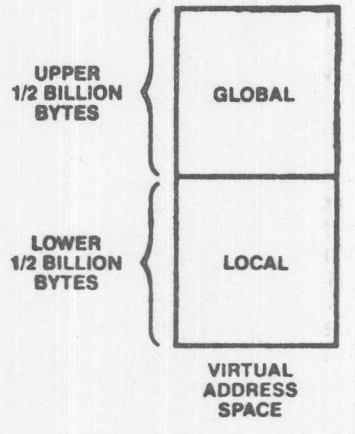


Figure 2. iAPX 286 User Virtual Address Space.

The global space is a virtual address space common to and shared by all users or tasks in the system. It typically contains the operating system or interfaces to operating system functions that are separate tasks. This typically includes system wide O.S. software like the kernel, real time interrupt handles, O.S. services available to users or tasks, and shared code and data like procedure libraries and task communication buffers.

The local space is totally private to the user or task and contains application code and data. From the users perspective, the global virtual address space makes the O.S. and all shared code and data a part of his address space. From the operating system perspective the user's local private address space is also a direct part of the O.S. address space allowing the O.S. quick direct access to the user's data. The O.S. can use this perspective to its advantage. Knowing that the local address space is private to the task, the O.S. can maintain O.S. related task specific data tables like memory allocation maps, I/O allocations, etc., within the local address space. This becomes particularly advantageous when swapping users in and out of a multi-user system since all task or user specific information is contained within the user or task's private address space.

Figure 3 shows how this mechanism applies to a system with multiple users or multiple tasks. Each task or user has a separate and private local virtual address space inaccessible from any other user's local virtual address space. At any instance in time, one user or task is executing. During that instance, the virtual address space of the system consists of the global address space and that user or task's private local address space. This technique allows all users to have private memory fully protected from all other users or tasks while sharing a common address space containing the O.S. and shared code and data. Making the O.S. part of the user's address space makes access to system procedures simple and straightforward. Only a standard call instruction is necessary for the user to transfer control to an O.S. service. Since the user address space is likewise within the O.S. address space, the O.S. has direct and simple access to user data areas, I/O buffers, etc. Figure 4 demonstrates how the mechanisms of the iAPX 286 would be applied to the typical system environment discussed at the beginning of this paper.

The memory model also satisfies the need for protection of users from each other and dynamic relocatability of code and data in addition to a clean user/O.S. interface. The actual mapping of global and local virtual address spaces to physical memory is shown in Figure 5. The keys to address mapping in the iAPX 286 are descriptor tables. The global address space and each local address space are described by separate descriptor tables. The upper 13 bits of a 32 bit virtual address are used as an index into either the local or global descriptor table. One bit of the virtual address also selects either the global or local virtual address spaces. The selected descriptor table entry contains the base address of the requested code or data, a limit value and access rights. The final physical address is formed by adding the 16 bit offset value of the virtual address to the base address contained in the descriptor. This process is shown in Figure 6. Since descriptor tables are memory resident, every time a descriptor is

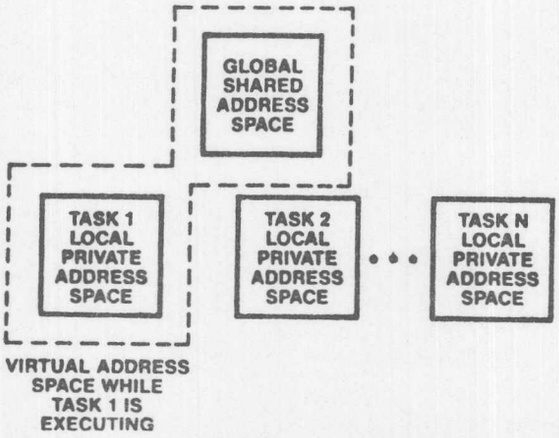


Figure 3. Conceptual View of Multi-user or Multitasking System Virtual Address Spaces.

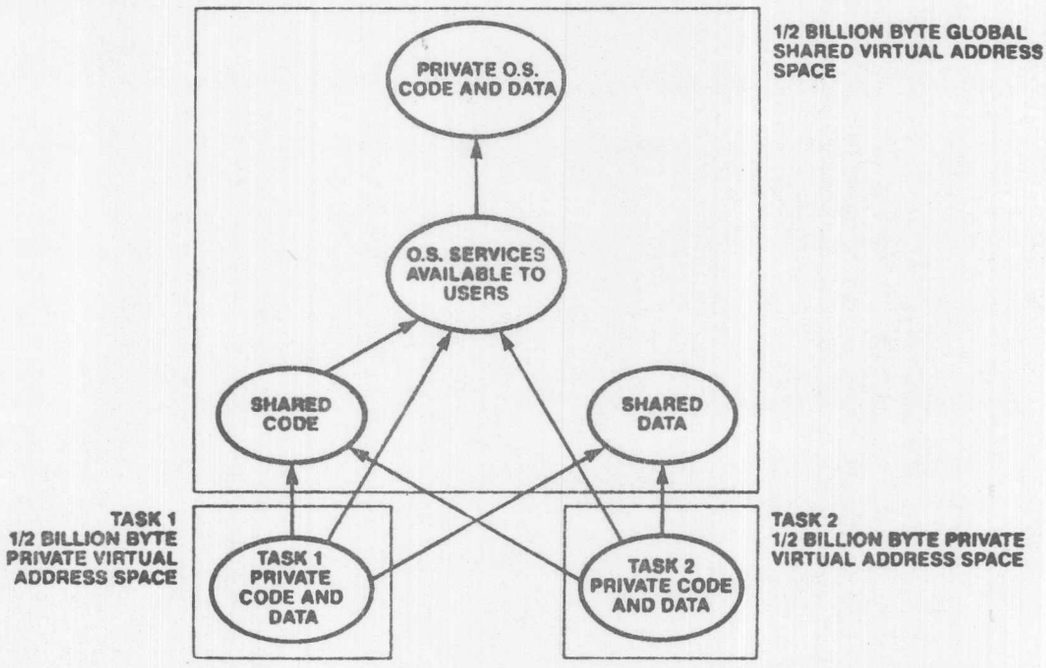


Figure 4. iAPX 286 Applied to Multi-user, Multitasking System Needs.

accessed, it is automatically cached into a chip descriptor cache where it is held for future use. This prevents subsequent access to the same area of code or data from requiring access to the memory based information. In addition to the base and limit fields, the descriptor also contains a present bit and accessed bit for virtual memory swapping and access rights bits which define usage rights like read only, execute only, etc.

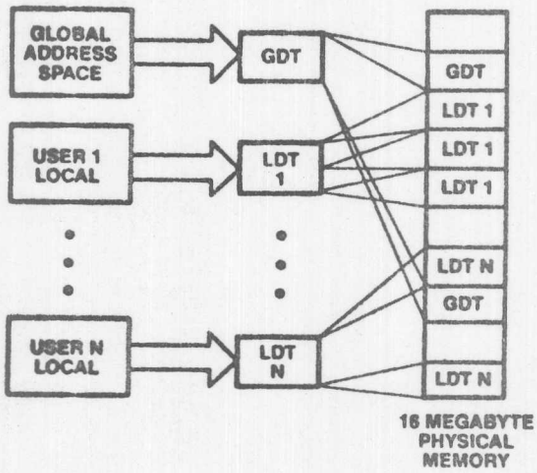


Figure 5. Virtual to Physical Address Translation.

The global and currently accessable local descriptor table locations in memory are defined by special registers internal to the iAPX 286. Since the global address space remains constant, this register is typically only loaded as part of system initialization. The register pointing to the currently active local descriptor table is reloaded with the address of the new table every time the system switches to a new task or user. This register is not accessable to the task or user and guarantees that during execution, the task or user's address space is uniquely defined by the global descriptor table and user or task's own local descriptor table. Additional checks like index limit checks to prevent accessing a descriptor beyond the end of the descriptor table and offset limit checks to prevent accessing outside the area of memory defined by a descriptor guarantee total isolation of a user's or task's address space from other users or tasks.

The advantages of the system software and user software sharing a common virtual address space are many. The O.S. has clean and simple access to user data areas, the user maintains a simple call interface to the O.S. and access to interrupt service routines is direct and fast.

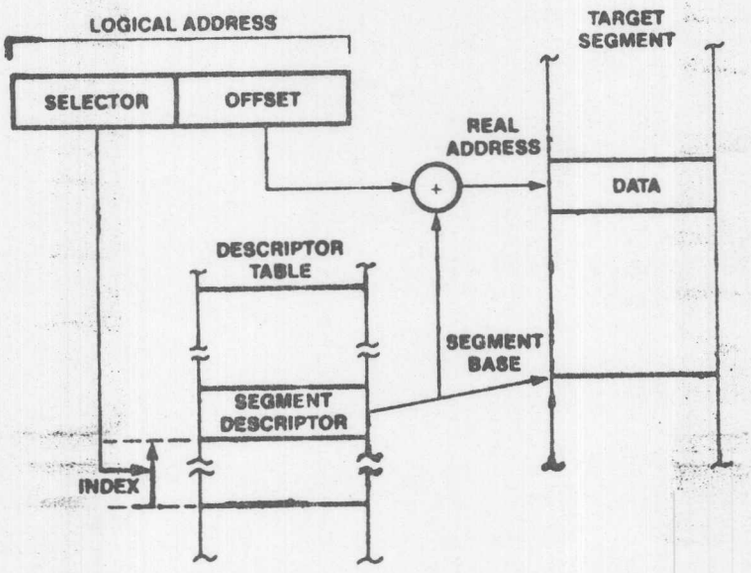


Figure 6. iAPX·286 Virtual Address Translation.

By maintaining high frequency and time critical interrupt service routines in the global address space, they are always immediately accessable regardless of which user or task is currently executing. In the iAPX 286, this results in an interrupt response time of under four microseconds with an eight megahertz CPU with full memory management and protection.

Virtual memory is supported in the system implementation via a combination of CPU facilities and software. The basic facilities provided by the iAPX 286 include the present bit of the descriptor which automatically forces an exception interrupt if a not-present segment of the user's virtual address space is accessed, an accessed bit for usage profiling by software implementing the swapping policy, and full restartability of all instructions which use full virtual addresses. These facilities provide the required capabilities for development of a full virtual memory system without imposing restrictions or requirements on the system implementation.

While the points discussed above address the needs of user isolation, virtual memory and efficient user/O.S. interface, a subsequent dimension of protection is necessary to provide O.S./user isolation even though at any instance, the O.S. and currently active user or task share the same virtual address space. This need is illustrated in Figure 7 where task or user code or shared code like a library procedure attempt to access O.S. routines that should be restricted from access outside the O.S.

To accommodate the need for this inner task protection, the iAPX 286 provides a four level hierarchy of trust within each task. This four level mechanism effectively overlays the virtual address space to provide protection within the address space. Each area of code and data is assigned to one of four privilege levels which controls the ability of code to access other code and data and the accessibility of data by code. The inner most level as shown in Figure 8 is the most trusted code which can access any data within the global and local address spaces and is most protected from access by software at less privileged levels. The protection implies restricted manipulation of data and restricted or controlled transfer of control to code at a more privileged level. Subsequent levels in the hierarchy are protected from access by software at lower levels but not from access from more trusted levels. At any instant, the active task or user is executing within one of the four privilege levels. At that

Required to Protect Code and Data From Corruption
By Less Reliable Software in the Same Virtual Address Space

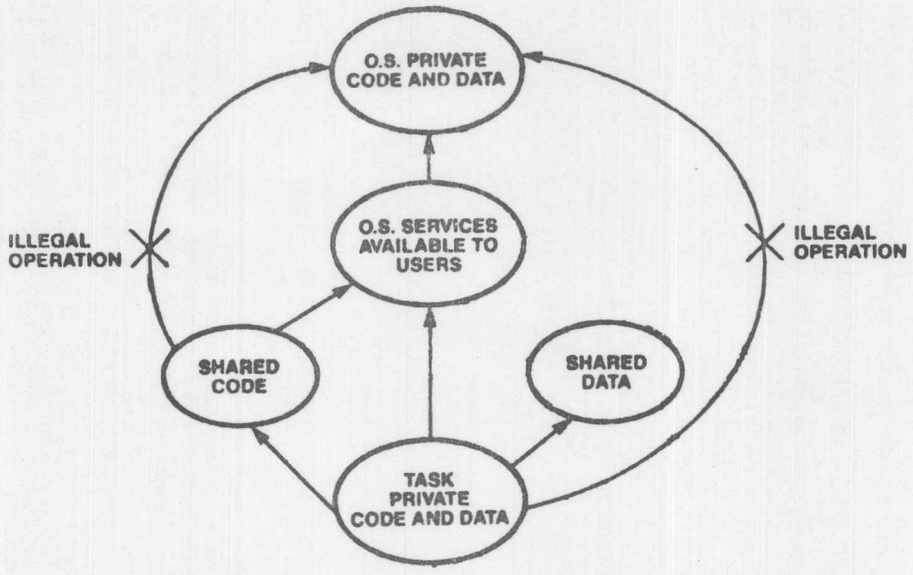


Figure 7. Inner Task Protection.

instance, the program may access data only at the same or a less privileged level. The program may transfer control anywhere directly within the same privilege level or to procedures at more privileged levels via a programmer transparent special control mechanism called a gate. A program cannot request service of (transfer control to) a procedure at a less privileged level since that procedure is considered less trusted (of questionable integrity) and may not be able to access data associated with the calling procedure at a higher level.

- A Hierarchy of Trust -

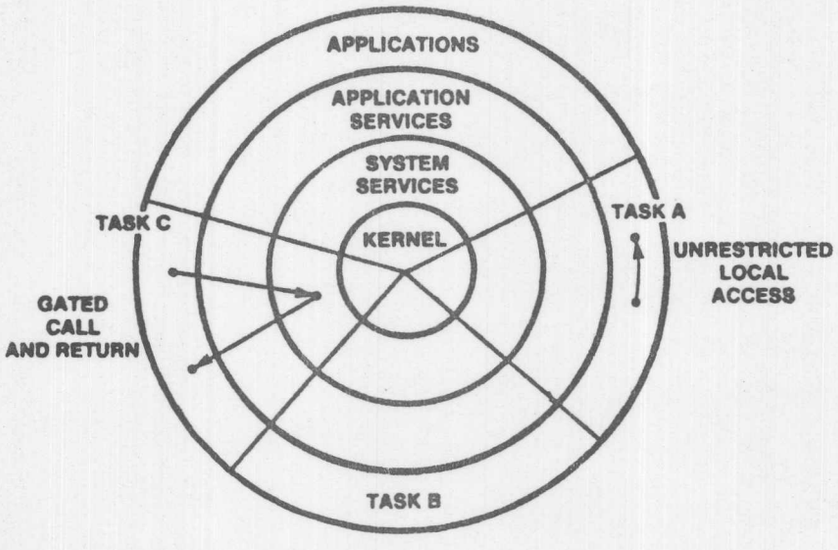


Figure 8. iAPX 286 Inner Task Protection Implementation.

This mechanism allows the system designer to partition the software and data structures among the various privilege levels for maximum system integrity. Privilege level 0, the most trusted level is typically reserved for the system kernel, level 1 for the O.S. services, level 2 for library procedures and custom O.S. extensions while level 3 is typically assigned for task and user application programs and data.

An example of data access control is shown in Figure 9. Code executing at privilege level two can access data at levels two and three but not at levels one and zero. An example of code access control is shown in Figure 10. Here the code at level two can use gates at levels two and three to transfer control to code at higher levels but cannot use gates at higher levels or directly transfer control to code at more privileged levels.

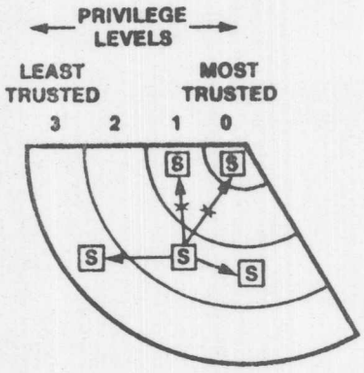


Figure 9. Data Access Control.

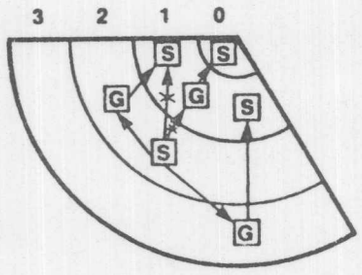


Figure 10. Code Access Control.

The basic operation of the gate mechanism is shown in Figure 11. The gate is contained in the descriptor table like a code segment, but is an additional level of indirection between the calling program and the target destination. If the virtual address specified in the call selects a gate, the iAPX 286 checks that the gate is at the same or less privileged level than the caller. If it is, a new virtual address contained within the gate is used to specify the actual descriptor of the target code segment and the entry point to the target code segments. The target code segment can be at the same or more privileged level, but not less privileged.

This mechanism controls the visibility of more privileged code from less privileged code. If the caller were to directly specify the descriptor of the more privileged routine, a protection check would be invoked. Therefore, the code is only accessable to the caller through the gate. If a gate does not exist at the callers privilege level or a less trusted privilege level, the caller cannot transfer control to the code. However, gates may exist at more privileged levels so more privileged code (for example at level 1) could access the code (for example at level 0).

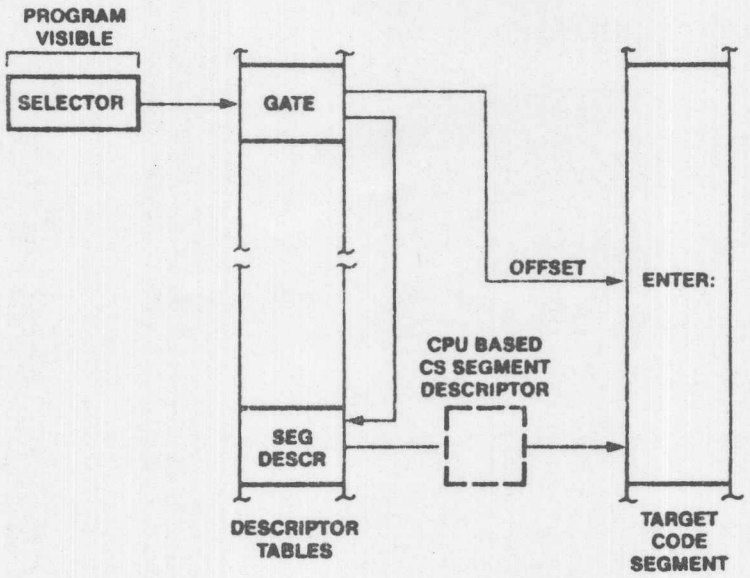


Figure 11. Control Transfer Via Gates.

In addition to controlling access to more privileged code, gates also serve another important functon. To prevent the user or task software from disrupting the stack for more privileged software, each privilege level is assigned a unique and separate stack. Therefore, when less privileged code calls more trusted software, it pushes parameters onto its stack and calls via the gate.

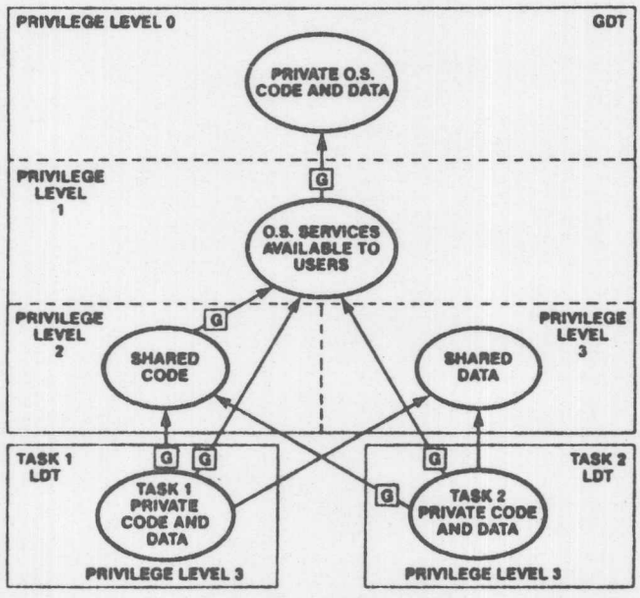
To prevent more privileged software from requiring access to the caller's stack, the gate contains a count of the number of parameters to be copied from the caller's stack to the called procedure's stack. The copy process is automatically performed by the iAPX 286 during the call. The resulting stack image seen by the called procedure is the same as the calling procedure. If a stack error occurs, it effectively occurs in the user's privilege level and not in the O.S. This capability significantly improves system call performance and simplifies the system software by eliminating the need to access stacks at various levels with defensive software.

An associated protection capability in the iAPX 286 is the support for verification of address parameters passed by the caller. Special instruction in the CPU allow the called procedure to restrict all addresses passed on the stack to the privilege level of the caller. This prevents an errant program from passing an address to the O.S.'s own data tables and structures to an O.S. procedure for manipulation and possible distraction on behalf of a user or task program. Although all addresses can be restricted, an invalid address will not cause an exception fault unless an attempt is made to use it to allow for dummy variables.

Also of critical importance to multi-user and multitasking systems in a protection environment is the system performance in task switching. To address this need, the iAPX 286 provides a fully integrated task switch capability. An extension of the gate mechanism, the facility allows a program to call or jump to (invoke) another task without operating system intervention. When used, the mechanism automatically saves the entire state of the current task, loads the state of the new task and transfers control to that task. This includes switching to the new users virtual address space and protection attributes. The entire operation requires only 22 microseconds with an eight

megahertz CPU. Back linkages are also automatically maintained for nested task invocations.

Figure 12 demonstrates how the iAPX 286 privilege levels and gates might be applied to completely satisfy the memory management and protection requirements of a multi-user and multitasking system. The result is a flexible and comprehensive system environment without sacrificing the full system performance.



ALL OTHER INTER-LEVEL ACCESSES OR CONTROL TRANSFERS ARE AUTOMATICALLY PROHIBITED

Figure 12. iAPX 286 Applied To Multi-user, Multitasking System Needs.